Customizing Pencil ±

writing your own path-, draw-, fill- and stroke- procedures

The code for all postscript procedures used in PENCIL is located in two files: "*pencilDefinitions.ps*" and "*pencilControlProcDefinitions.ps*". They must be in the same directory as Pencil.app. There is a third file, "*pencilMethodDescriptions*", which lists the method-names as they appear in the pop-up-list and the corresponding postscript-procedures (The syntax is obvious. Open the file in Edit and add new entries once you have added some new postscript-procedures). To try out a new path/draw/fill/stroke- method without changing the files, you can choose the "**Info>Custom PS...**" menu item and type the postscript code in the appearing panel. The new postscript-procedure will be available once you have clicked on "Send to PS Server". It will not appear in any of the pop-up lists, but you can type its name directly in the corresponding text-field. Your code will be saved along with the current document. **NOTE:** You must click on "Send to PS Server" to save the PS code in the current view's corresponding instance variable. Don't select another document until you have done this because otherwise you will lose your changes to the code.

APPLICATION-DEFINED PS-VARIABLES

In general, when one of your procedures is called, the following postscript-variables are defined by the application:

ctrl	array of control-point-coordinates in the form [x1 y1 x2 y2 x3 y3...], e.g. [0 .5 8 .3]
nctrl	number of controlpoints
udt	a procedure that, when executed, sets the user-defined-variables (actually, this is the postscript-code the user types in the "user variable definitions" text-field)
cl	a procedure that, when executed, will place r1 g1 b1 r2 g2 b2 on the stack, i. e. the rgb-values of user-defined colors 1 and 2.

PATH-METHODS

The purpose of a path-procedure is to construct a path taking into account the information given in ctrl (and perhaps udt). To loop over all controlpoints, use this code:

```
0 1 nctrl 1 sub { /i exch 2 mul def
        ctrl i get              % push x(i) on stack
        ctrl i 1 add get        % push y(i) on stack
                                        % do something
                        } for
```

Example

This is the code for the "Open Polygon" path-method:

```
/polygonO {
        nctrl 2 ge {
                ctrl 0 get ctrl 1 get moveto
                1 1 nctrl 1 sub { /i exch 2 mul def
                ctrl i get ctrl i 1 add get lineto } for
        } { 0 0 moveto } ifelse
} def
/polygonOcontrol { gencontrol } def
```

It tests whether there are at least two controlpoints. If yes, it constructs an open polygon by using a moveto on the first controlpoint and linetos to all other points. If no, it constructs an empty path by only calling moveto. NOTE: Always construct a path, at least an empty one (0 0 moveto). Other procedures rely on the existence of a current path.

For every path procedure "NAME" there has to be a procedure named "NAMEcontrol" that draws the controlpoints and additional lines etc. when the user has selected the path. Normally, it is enough to call "gencontrol" which will draw the controlpoints in black. These control procedures are defined in "pencilControlProcDefinitions.ps". They are not defined when printing. See "hermitecontrol" for an example of a special control-procedure.

FILL-METHODS

They are called by the draw-procedure with a current path already constructed. The simplest fill-procedure is the standard postscript "fill".

Example
This is the code for the "Grid" fill-method:

```
/gridsize 5 def
/grid {
udt
doClip
0 gridsize height
{ 0 exch moveto width 0 rlineto stroke } for
0 gridsize width
{ 0 moveto 0 height rlineto stroke } for
endClip
/gridsize 5 def
} def
```

At first, udt is called (this may change the value of gridsize). "doClip" clips to the current path, translates to the lower left corner of the path's bounding box and defines width and height. You should use doClip/endClip in the following way:

```
/myfill {
doClip
...            % fill rectangle (0,0)-(width,height)
endClip
} def
```

USER-DEFINED VARIABLES

In the "grid"-example, "gridsize" is reset to its default value after it has been used. When you want to use a user-defined variable, do it like this:

```
/myVar 10 def          % default value
/myProc {
udt                    % user-defined
...                    % use myVar
/myVar 10 def          % reset to default value
} def
```

STROKE-METHODS

They are called by the draw-procedure with a currentpath already constructed. The simplest stroke-procedure is the standard postscript "stroke".

Example
This is the code for the "Doubleline" stroke-method:

```
/doubleline {
	gsave currentlinewidth 2 mul setlinewidth stroke grestore
	1 setgray stroke
} def
```

It strokes the current path in the current color with double the current linewidth and then strokes it again in white using the current linewidth. The gsave/grestore-pair around the first stroke is necessary to avoid destroying the current path.

DRAW-METHODS

When a draw-method is called, the following variables are defined in addition to cl, udt, ctrl and nctrl:

cp	Current path-procedure
cfl	Current fill-procedure
cst	Current stroke-procedure

A draw-procedure uses cp, cfl and cst to draw the current path.

Example
This is the code for the "Filled&Stroked" draw-method:

```
/drawFS {
	cl setrgbcolor
	cp
```

```
        gsave
        cfl
        grestore
        setrgbcolor
        cst
} def
/drawFSBB { fsbb } def
```

It executes cl (r1 g1 b1   r2 g2 b2 are pushed on the stack), sets color 2 by calling setrgbcolor, constructs the current path by calling cp, fills it by calling cfl, sets the current color to color 1 and strokes the current path by calling cst.

Every draw-procedure "name" must have a "nameBB"-procedure that pushes the bounding box of the current path on the stack. There are three generally useful procedures for this purpose:

fbb	pushes the bounding box of the cp on the stack
fsbb	does the same but takes into account the current linewidth
enlbb	(dlx dly drx dry enlbb) takes the bounding box on the stack and enlarges it by changing its lower left and upper right corners by the specified amounts

(e. g. "fbb -5 -5 5 5 enlbb" enlarges the BB
5 pixels in every direction)

±±

A final example

± Start Pencil.app (If you have already done so, save and close your current document)
± Open a new document (Command-n)
± Open the CustomPS-panel (Command-2)
± Copy the following code into it:

```
/myfill {
doClip
0 20 width { /x exch def
0 20 height { /y exch def
x y 8 0 360 arc stroke
} for
} for
endClip
} def
```

± Click on "Send to PS Server"

± Draw a path
± Change the draw-method to "Filled"
± Click on "Expert settings" and enter "myfill" into the "Fill-method" textfield. Now you should see your fill-pattern.

±±
In the **SampleImages**-directory, there are some examples of custom-defined methods. Bring up the CustomPS-Panel (Command-2) and open these documents.

±±
If you have created stable methods that you believe to be generally useful and would like to have them included in the next version of Pencil, mail me your changes to pencilDefinitions.ps, pencilControlProcDefinitions.ps and pencilMethodDescriptions. (a0047@freenet.uni-bayreuth.de)
If you have created nice effects, consider submitting them to some ftp-server in the form of a Pencil-document (as "CustomPS"-methods). Perhaps someone will find them useful.